Table of Contents

Best Practices

Lustre Best Practices

Lustre filesystems are shared among many users and many application processes, which causes contention for various Lustre resources. This article explains how Lustre I/O works, and provides best practices fro improving application performance.

How does Lustre I/O work?

When a client (a compute node from your job) needs to create or access a file, the client queries the metadata server (MDS) and the metadata target (MDT) for the layout and location of the file's stripes. Once the file is opened and the client obtains the striping information, the MDS is no longer involved in the file I/O process. The client interacts directly with the object storage servers (OSSes) and object storage targets (OSTs) to perform the I/O operations such as locking, disk allocation, storage, and retrieval.

If multiple clients try to read and write the same part of a file at the same time, the Lustre distributed lock manager enforces coherency so that all clients see consistent results.

Jobs being run on Pleiades content for shared resources in NAS's Lustre filesystem. The Lustre server can only handle about 15,000 remote procedure calls (RPCs, inter-process communications that allow the client to cause a procedure to be executed on the server) per second. Contention slows the performance of your applications and weakens the overall health of the Lustre filesystem. To reduce contention and improve performance, please apply the examples below to your compute jobs, while working in our high-end computing environment.

Best Practices

- **Avoid using *ls -l***

  The *ls -l* command displays information such as ownership, permission and size of all files and directories. The information on ownership and permission metadata is stored on the MDTs. However, the file size metadata is only available from the OSTs. So, the *ls -l* command issues RPCs to the MDS/MDT and OSSes/OSTs for every file/directory to be listed. RPC requests to the OSSes/OSTs are very costly and can take a long time to complete for many files and directories.

  - Use *ls* by itself if you just want to see if a file exists.

  - Use *ls -l filename* if you want the long listing of a specific file.

- **Avoid having a large number of files in a single directory**

  Opening a file keeps a lock on the parent directory. When many files in the same directory are to be opened, it creates contention. It is better to split a huge number of files (in the thousands or more) into multiple sub-directories to minimize contention.

- **Avoid accessing small files on Lustre filesystems**

  Accessing small files on the Lustre filesystem is not efficient. If possible, keep them on an NFS-mounted filesystem (such as your home filesystem) or copy them from Lustre to /tmp on each node at the beginning of the job and access them from there.

- **Use a stripe count of 1 for directories with many small files**

  If you have to keep small files on Lustre, be aware that *stat* operations are more efficient if each small file resides in one OST. Create a directory to keep small files, set the stripe count to 1 so that only one OST will be needed for each file. This is useful when you extract source and header files (which are usually very small files) from a tarfile.

  ```
  pfe1% mkdir dir_name
  pfe1% lfs setstripe -s 1m -c 1 dir_name
  pfe1% cd dir_name
  pfe1% tar -xf tarfile
  ```

  If there are large files in the same directory tree, it may be better to allow them to stripe across more than one OST. You can create a new directory with a larger stripe count and copy the larger file to that directory. Note that moving files into that directory with the *mv* command will not change the strip count of the files. Files must be created in or copied to a directory to inherit the stripe count properties of a directory.

  ```
  pfe1% mkdir dir_count_4
  pfe1% lfs setstripe -s 1m -c 4 dir_count_4
  pfe1% cp file_count_1 dir_count_4
  ```

  If you have a directory with many small files (less than 100MB) and a few very large files (greater than 1GB), then it may be better to create a new subdirectory with a larger stripe count. Store just the large files and create symbolic links to the large files using the *symlink* command.

  ```
  pfe1%  mkdir bigstripe
  pfe1%  lfs setstripe -c 16 -s 4m bigstripe
  pfe1%  ln -s bigstripe/large_file  large_file
  ```

- **Use mtar for creating or extracting a tar file**

  A modified gnu tar command, */usr/local/bin/mtar*, is Lustre stripe aware and will create tar files or extract files with appropriately sized stripe counts. Currently, the number of streps is set to the number of gigabytes of the file.

- **Keep copies of your source on the Pleiades home filesystem and/or Lou**

  Be aware that files under /nobackup[p1,p2,p10-p60] are not backed up. Make sure that you have copies of your source codes, makefiles, and any other important files saved on your Pleiades home filesystem or on Lou, the NAS storage system.

- **Avoid accessing executables on Lustre filesystems**

  There have been a few incidents on Pleiades where users' jobs encountered problems while accessing their executables on /nobackup. The main issue is that the Lustre clients can become unmounted temporarily when there is a very high load on the Lustre filesystem. This can cause a bus error when a job tries to bring the next set of instructions from the inaccessible executable into memory.

  Executables run slower when run from the Lustre filesystem. It is best to run executables from your home filesystem on Pleiades. On rare occasions, running executables from the Lustre filesystem can cause executables to be corrupted. Avoid copying new executable over existing executables of the same within the Lustre filesystem. The copy causes a window of time (about 20 minutes) where the executable will not function. Instead, the executable should be accessed from your home filesystem during runtime.
- **Increase the stripe_count for parallel writes to the same file**

  When multiple processes are writing blocks of data to the same file in parallel, I/O performance is better for large files when the stripe_count is set to a larger value. The stripe count sets the number of OSTs the file will be written to. By default, the stripe count is set to 1. While this default setting provides for efficient access of metadata for example to support "ls -l"&emdash;large files should use stripe counts of greater than 1. This will increase the aggregate I/O bandwidth by using multiple OSTs in parallel instead of just one. A rule of thumb is to use a stripe count approximately equal to the number of gigabytes in the file.

  It is also better to make the stripe count be an integral factor of the number of processes performing the write in parallel so that one achieves load balance among the OSTs. For example, set the stripe count to 16 instead of 15 when you have 64 processes performing the writes.

- **Limit the number of processes performing parallel I/O**

  Given that the numbers of OSSes and OSTs on Pleiades are about a hundred or fewer, there will be contention if a huge number of processes of an application are involved in parallel I/O. Instead of allowing all processes to do the I/O, choose just a few processes to do the work. For writes, these few processes should collect the data from other processes before the writes. For reads, these few processes should read the data and then broadcast the data to others.

- **Stripe align I/O requests to minimize contention**

  Stripe aligning means that the processes access files at offsets that correspond to stripe boundaries. This helps to minimize the number of OSTs a process must communicate for each I/O request. It also helps to decrease the probability that multiple processes accessing the same file communicate with the same OST at the same time.

  One way to stripe-align a file is to make the stripe size the same as the amount of data in the write operations of the program.

- **Avoid repetitive *stat* operations**

  Some users have implemented logic in their scripts to test for the existence of certain files. Such tests generate *stat* requests to the Lustre server. When the testing becomes excessive, it creates a significant load on the filesystem. A workaround is to slow down the testing by adding *sleep* in the logic. For example, the following user script tests the existence of the files WAIT and STOP to decide what to do next.

```
touch WAIT
 rm STOP

 while ( 0 <= 1  )
  if(-e WAIT) then
    mpiexec ...
    rm WAIT
  endif
  if(-e STOP) then
    exit
  endif
 end
```

  When neither the WAIT nor STOP file exists, the loop ends up testing for their existence as fast as possible (on the order of 5000 times per second). Adding a *sleep* inside the loop slows down the testing.

```
touch WAIT
 rm STOP

 while ( 0 <= 1  )
  if(-e WAIT) then
    mpiexec ...
    rm WAIT
  endif
```

```
  if(-e STOP) then
    exit
  endif
  sleep 15
end
```

- **Avoid multiple processes opening the same file(s) at the same time**

  On Lustre filesystems, if multiple processes try to open the same file(s), some processes will not able to find the file(s) and the job will fail.

  The source code can be modified to call the sleep function between I/O operations. This will reduce the occcurence of multiple access attempts to the same file from different processes simultaneously.

```
100   open(unit,file='filename',IOSTAT=ierr)
      if (ierr.ne.0) then
       ...
      call sleep(1)
      go to 100
      endif
```

  When opening a read-only file in Fortran, use ACTION='read' instead of the default ACTION='readwrite'. The former will reduce contention by not locking the file.

```
open(unit,file='filename',ACTION='READ',IOSTAT=ierr)
```

- **Avoid repetitive open/close operations**

  Opening files and closing files incur overhead and repetitive open/close should be avoided.

  If you intend to open the files for read only, make sure to use **ACTION='READ'** in the open statement. If possible, read the files once each and save the results, instead of reading the files repeatedly.

  If you intend to write to a file many times during a run, open the file once at the beginning of the run. When all writes are done, close the file at the end of the run.

Reporting Problems

If you report performance problems with a Lustre filesystem, please be sure to include the time, hostname, PBS job number, name of the filesystem, and the path of the directory or file that you are trying to access.Your report will help us correlated issues with recorded performance data to determine the cause of efficiency problems.

Streamlining File Transfers from Pleiades Compute Nodes to Lou

Some users prefer to streamline the storage of files (created during a job run) to Lou, within a PBS job. Since Pleiades compute nodes do not have network access to the outside world, all file transfers to Lou within a PBS job must go through the front-ends (pfe[1-12], bridge[1,2]) first.

Here is an example of what you can add to your PBS script to accomplish this:

1. Ssh to a front-end node (for example, bridge2) and create a directory on Lou where the files are to be copied.

   ```
   ssh -q bridge2 "ssh -q lou mkdir -p $SAVDIR"
   ```
   Here, $SAVDIR is assumed to have been defined earlier in the PBS script. Note the use of *-q* for quiet-mode, and double quotes so that shell variables are expanded prior to the *ssh* command being issued.

2. Use scp via bridge[1,2] to transfer the files.

   ```
   ssh -q bridge2 "scp -q $RUNDIR/* lou:$SAVDIR"
   ```
   Here, $RUNDIR is assumed to have been defined earlier in the PBS script.

Avoiding Job Failure from Overfilling /PBS/spool

Before a PBS job is completed, its error and output files are kept in the /PBS/spool directory of the first node of your PBS job. The space under /PBS/spool is limited, however, and when it fills up, any job that tries to write to /PBS/spool may die. To prevent this, you should *not* write large amount of contents in the PBS output/error files.

If your executable normally produces a lot of output to the screen, you should redirect its output in your PBS script. For example:

```
#PBS ...
mpiexec a.out > output
```

To see the contents of your PBS output/error files before your job completes, follow the two steps below:

1. Find out the first node of your PBS job using "-W o=+rank0" for qstat:

   ```
   %qstat -u your_username -W o=+rank0
   JobID         User   Queue  Jobname   TSK Nds    wallt S    wallt  Eff Rank0
   ------------- ------ ------ -------- ---- --- -------- - -------- ---- ---------
   868819.pbspl1 zsmith long   ABC       512  64 5d+00:00 R 3d+08:39 100% r162i0n14
   ```

   This shows that the first node is r162i0n14.

2. Log in to the first node and *cd* to /PBS/spool to find your PBS stderr/out file(s). You can view the content of these files using *vi* or *view*.

```
%ssh r162i0n14
%cd /PBS/spool
%ls -lrt
-rw------- 1 zsmith a0800 49224236 Aug  2 19:33 868819.pbspl1.nas.nasa.gov.OU
-rw------- 1 zsmith a0800  1234236 Aug  2 19:33 868819.pbspl1.nas.nasa.gov.ER
```